

# TDRV020-SW-82

## Linux Device Driver

Reconfigurable FPGA

Version 1.1.x

## User Manual

Issue 1.1.1

October 2024

## TDRV020-SW-82

Linux Device Driver

Reconfigurable FPGA

Supported Modules:

TMPE623

TMPE627

TMPE633

This document contains information, which is proprietary to TEWS Technologies GmbH. Any reproduction without written permission is forbidden.

TEWS Technologies GmbH has made any effort to ensure that this manual is accurate and complete. However TEWS Technologies GmbH reserves the right to change the product described in this document at any time without notice.

TEWS Technologies GmbH is not liable for any damage arising out of the application or use of the device described herein.

©2017-2024 by TEWS Technologies GmbH

<b>Issue</b>	<b>Description</b>	<b>Date</b>
1.0.0	First Issue	December 20, 2017
1.1.0	RestorePciHeader function added	January 25, 2018
1.1.1	New Address of TEWS Technologies GmbH	October 8, 2024

# Table of Contents

<b>1</b>	<b>INTRODUCTION.....</b>	<b>4</b>
<b>2</b>	<b>INSTALLATION.....</b>	<b>5</b>
	2.1 Build and install the Device Driver .....	5
	2.2 Uninstall the Device Driver .....	6
	2.3 Install Device Driver into the running Kernel.....	6
	2.4 Remove Device Driver from the running Kernel.....	6
	2.5 Change Major Device Number .....	7
<b>3</b>	<b>API DOCUMENTATION .....</b>	<b>8</b>
	<b>3.1 General Functions.....</b>	<b>8</b>
	3.1.1 tdrv020Open .....	8
	3.1.2 tdrv020Close .....	10
	3.1.3 tdrv020GetPciInfo .....	12
	3.1.4 tdrv020RestorePciHeader.....	14
	<b>3.2 Device Access Functions.....</b>	<b>16</b>
	3.2.1 tdrv020Read8 .....	16
	3.2.2 tdrv020ReadBE16.....	19
	3.2.3 tdrv020ReadLE16 .....	22
	3.2.4 tdrv020ReadBE32.....	25
	3.2.5 tdrv020ReadLE32 .....	28
	3.2.6 tdrv020Write8.....	31
	3.2.7 tdrv020WriteBE16 .....	34
	3.2.8 tdrv020WriteLE16 .....	37
	3.2.9 tdrv020WriteBE32.....	40
	3.2.10 tdrv020WriteLE32 .....	43
	<b>3.3 Resource Mapping Functions.....</b>	<b>46</b>
	3.3.1 tdrv020PciResourceMap.....	46
	3.3.2 tdrv020PciResourceUnmap .....	48
	<b>3.4 Interrupt Functions .....</b>	<b>50</b>
	3.4.1 tdrv020InterruptConfig .....	50
	3.4.2 tdrv020InterruptWait .....	55
	3.4.3 tdrv020InterruptRegisterCallbackThread.....	58
	3.4.4 tdrv020InterruptUnregisterCallback .....	63
	<b>3.5 Endian Conversion Functions .....</b>	<b>65</b>
	3.5.1 endian_be16 .....	65
	3.5.2 endian_le16 .....	66
	3.5.3 endian_be32 .....	67
	3.5.4 endian_le32 .....	68
<b>4</b>	<b>DIAGNOSTIC.....</b>	<b>69</b>

# 1 Introduction

The TDRV020-SW-82 Linux device driver allows the operation of the TDRV020 compatible devices conforming to the Linux I/O system specification.

The TDRV020-SW-82 device driver was designed to provide register access to an FPGA application.

The TDRV020-SW-82 device driver supports the following features:

- Read/write access to FPGA registers (8,16,32-bit)
- Resource allocation for supported modules
- Wait for interrupts
- Register Callback functions for interrupt handling
- Driver functions are thread-safe as long as unique handles are used.

The TDRV020-SW-82 supports the modules listed below:

TMPE623	Reconfigurable FPGA with Digital I/O	PCIe Mini Card
TMPE627	Reconfigurable FPGA with AD/DA and Digital I/O	PCIe Mini Card
TMPE633	Reconfigurable FPGA with Digital I/O	PCIe Mini Card

**In this document all supported modules and devices will be called TDRV020. Specials for a certain device will be advised.**

To get more information about the features and use of supported devices it is recommended to read the manuals listed below.

Corresponding Hardware User manual
Related FPGA Development Kit (FDK) documentation

## 2 Installation

The directory TDRV020-SW-82 on the distribution media contains the following files:

TDRV020-SW-82-1.1.1.pdf	This manual in PDF format
TDRV020-SW-82-SRC.tar.gz	GZIP compressed archive with driver source code
ChangeLog.txt	Release history
Release.txt	Information about the Device Driver Release

The GZIP compressed archive TDRV020-SW-82-SRC.tar.gz contains the following files and directories:

Directory path 'tdrv020':

tdrv020.c	Driver source code
tdrv020def.h	Driver include file
tdrv020.h	Driver include file for application program
Makefile	Device driver make file
makenode	Script for device node creation
COPYING	Copy of the GNU Public License (GPL)
api/tdrv020api.h	API include file
api/tdrv020api.c	API source file
example/tdrv020exa.c	Example application
example/Makefile	Example application makefile
include/config.h	Driver independent library header file
include/tpmodule.c	Driver independent library
include/tpmodule.h	Driver independent library header file
include/tpxxxhwdep.h	HAL library header file
include/tpxxxhwdep.c	HAL library source file

In order to perform an installation, extract all files of the archive TDRV020-SW-82-SRC.tar.gz to the desired target directory. The command 'tar -xzvf TDRV020-SW-82-SRC.tar.gz' will extract the files into the local directory.

- Login as *root* and change to the target directory
- Copy tdrv020.h to */usr/include*

### 2.1 Build and install the Device Driver

- Login as *root*
- Change to the target directory
- To create and install the driver in the module directory */lib/modules/<version>/misc* enter:
  - # make install**
- To update the device driver's module dependencies, enter:
  - # depmod -aq**

## 2.2 Uninstall the Device Driver

- Login as *root*
- Change to the target directory
- To remove the driver from the module directory */lib/modules/<version>/misc* enter:  
**# make uninstall**

## 2.3 Install Device Driver into the running Kernel

- To load the device driver into the running kernel, login as root and execute the following commands:  
**# modprobe tdrv020drv**
- After the first build or if you are using dynamic major device allocation it is necessary to create new device nodes on the file system. Please execute the script file *makenode* to do this. If your kernel has enabled a device file system (devfs or sysfs with udev) then you have to skip running the *makenode* script. Instead of creating device nodes from the script the driver itself takes creating and destroying of device nodes in its responsibility.  
**# sh makenode**

On success the device driver will create a minor device for each TDRV020 device found. The first TDRV020 device can be accessed with device node */dev/tdrv020\_0*, the second module with device node */dev/tdrv020\_1* and so on.

The assignment of device nodes to physical TDRV020 modules depends on the search order of the PCI bus driver.

## 2.4 Remove Device Driver from the running Kernel

- To remove the device driver from the running kernel login as root and execute the following command:  
**# modprobe -r tdrv020drv**

If your kernel has enabled devfs or sysfs (udev), all */dev/tdrv020\_x* nodes will be automatically removed from your file system after this.

**Be sure that the driver isn't opened by any application program. If opened you will get the response "*tdrv020drv: Device or resource busy*" and the driver will still remain in the system until you close all opened files and execute *modprobe -r* again.**

## 2.5 Change Major Device Number

This paragraph is only for Linux kernels without dynamic device file system installed. The TDRV020 driver uses dynamic allocation of major device numbers per default. If this isn't suitable for the application it is possible to define a major number for the driver.

To change the major number, edit the file `tdrv020def.h`, change the following symbol to appropriate value, and enter `make install` to create a new driver.

TDRV020_MAJOR	Valid numbers are in range between 0 and 255. A value of 0 means dynamic number allocation.
---------------	---

### Example:

```
#define TDRV020_MAJOR 122
```

**Be sure that the desired major number isn't used by other drivers. Please check `/proc/devices` to see which numbers are free.**

## **3 API Documentation**

### **3.1 General Functions**

#### **3.1.1 tdrv020Open**

##### **NAME**

tdrv020Open – open a device.

##### **SYNOPSIS**

```
TDRV020_HANDLE tdrv020Open  
(  
    char      *DeviceName  
)
```

##### **DESCRIPTION**

Before I/O can be performed to a device, a device handle must be opened by a call to this function.

**The tdrv020Open function can be called multiple times (e.g. in different tasks).**

##### **PARAMETERS**

###### *DeviceName*

This parameter points to a null-terminated string that specifies the name of the device. The first TDRV020 device is named “/dev/tdrv020\_0” the second device is named “/dev/tdrv020\_1” and so on.



---

## EXAMPLE

```
#include "tdrv020api.h"

TDRV020_HANDLE hdl;

/*
** open the specified device
*/
hdl = tdrv020Open("/dev/tdrv020_0");
if (hdl == NULL)
{
    /* handle open error */
}
```

## RETURNS

A device handle, or NULL if the function fails. An error code will be stored in *errno*.

## ERROR CODES

The error codes are stored in *errno*.

The error code is a standard error code set by the I/O system.

## 3.1.2 tdrv020Close

### NAME

tdrv020Close – close a device.

### SYNOPSIS

```
TDRV020_STATUS tdrv020Close
(
    TDRV020_HANDLE    hdl
)
```

### DESCRIPTION

This function closes a previously opened device.

### PARAMETERS

*hdl*

This value specifies the device handle to the hardware module retrieved by a call to the corresponding open-function.

### EXAMPLE

```
#include "tdrv020api.h"

TDRV020_HANDLE    hdl;
TDRV020_STATUS    result;

/*
** close the device
*/
result = tdrv020Close(hdl);
if (result != TDRV020_OK)
{
    /* handle close error */
}
```

## RETURNS

On success, TDRV020\_OK is returned. In the case of an error, the appropriate error code is returned by the function.

## ERROR CODES

Error Code	Description
TDRV020_ERR_INVALID_HANDLE	The specified device handle is invalid

### 3.1.3 tdrv020GetPciInfo

#### NAME

tdrv020GetPciInfo – get information of the module PCI header

#### SYNOPSIS

```
TDRV020_STATUS tdrv020GetPciInfo
(
    TDRV020_HANDLE          hdl,
    TDRV020_PCIINFO_BUF    *pPciInfoBuf
)
```

#### DESCRIPTION

This function returns information of the module PCI header in the provided data buffer.

#### PARAMETERS

*hdl*

This argument specifies the device handle to the hardware module retrieved by a call to the corresponding open-function.

*pPciInfoBuf*

This argument is a pointer to the structure TDRV020\_PCIINFO\_BUF that receives information of the module PCI header.

```
typedef struct
{
    unsigned short    vendorId;
    unsigned short    deviceId;
    unsigned short    subSystemId;
    unsigned short    subSystemVendorId;
    int               pciBusNo;
    int               pciDevNo;
    int               pciFuncNo;
} TDRV020_PCIINFO_BUF;
```

*vendorId*

PCI module vendor ID.

*deviceId*

PCI module device ID

*subSystemId*  
PCI module sub system ID

*subSystemVendorId*  
PCI module sub system vendor ID

*pciBusNo*  
Number of the PCI bus, where the module resides.

*pciDevNo*  
PCI device number

*pciFuncNo*  
PCI function number

## EXAMPLE

```
#include "tdrv020api.h"

TDRV020_HANDLE      hdl;
TDRV020_STATUS      result;
TDRV020_PCIINFO_BUF pciInfoBuf

/*
** get module PCI information
*/
result = tdrv020GetPciInfo(hdl, &pciInfoBuf);

if (result != TDRV020_OK)
{
    /* handle error */
}
```

## RETURN VALUE

On success, TDRV020\_OK is returned. In the case of an error, the appropriate error code is returned by the function.

## ERROR CODES

Error Code	Description
TDRV020_ERR_INVALID_HANDLE	The specified device handle is invalid

### 3.1.4 tdrv020RestorePciHeader

#### NAME

tdrv020RestorePciHeader – Restore the PCI header of the User-FPGA

#### SYNOPSIS

```
TDRV020_STATUS tdrv020RestorePciHeader
(
    TDRV020_HANDLE    hdl
)
```

#### DESCRIPTION

This function restores the PCI header of the User-FPGA, using values stored upon driver start. After reconfiguration of a User-FPGA, the PCI header configuration is lost. To allow further accesses to the User-FPGA, the PCI header must be restored using this function.

#### PARAMETERS

*hdl*

This argument specifies the device handle to the hardware module retrieved by a call to the corresponding open-function.

#### EXAMPLE

```
#include "tdrv020api.h"

TDRV020_HANDLE    hdl;
TDRV020_STATUS    result;

/*
** Restore the PCI header of the User-FPGA
*/
result = tdrv020RestorePciHeader( hdl );
if (result != TDRV020_OK)
{
    /* handle error */
}
```

---

## RETURNS

On success, TDRV020\_OK is returned. In the case of an error, the appropriate error code is returned by the function.

## ERROR CODES

Error Code	Description
TDRV020_ERR_INVALID_HANDLE	The specified TDRV020_HANDLE is invalid.
TDRV020_ERR_NOSYS	This function is not supported by the device.

## 3.2 Device Access Functions

### 3.2.1 tdrv020Read8

#### NAME

tdrv020Read8 – read 8-bit values from PCI BAR space

#### SYNOPSIS

```
TDRV020_STATUS tdrv020Read8
(
    TDRV020_HANDLE    hdl,
    int               pciResource,
    int               offset,
    int               numItems,
    unsigned char     *pData
)
```

#### DESCRIPTION

This function reads the specified number of items from the PCI BAR space by using single byte (8-bit) accesses.

#### PARAMETERS

*hdl*

This argument specifies the device handle to the hardware module retrieved by a call to the corresponding open-function.

*pciResource*

This parameter specifies the desired PCI Memory resource to be used for this access. In general, a PCI target supports up to six base address registers. Following values are possible:

Value	Description
TDRV020_RES_MEM_1	First found PCI Memory area.
TDRV020_RES_MEM_2	Second found PCI Memory area.
TDRV020_RES_MEM_3	Third found PCI Memory area.
TDRV020_RES_MEM_4	Fourth found PCI Memory area.
TDRV020_RES_MEM_5	Fifth found PCI Memory area.
TDRV020_RES_MEM_6	Sixth found PCI Memory area.

The Base Address Register usage is programmable and can be changed by modifying the PCIe bridge configuration. Therefore the following table is just an example how the PCI Base Address Registers could be used.



PCI Base Address Register	PCI Address-Type	TDRV020 Resource
0	MEM	TDRV020_RES_MEM_1
1	MEM ( <i>not used</i> )	TDRV020_RES_MEM_2
2	MEM ( <i>not used</i> )	TDRV020_RES_MEM_3

*offset*

This argument specifies the start offset within the PCI BAR space.

*numItems*

This argument specifies the number of items (8-bit) to read.

*pData*

This argument is a pointer to an unsigned char buffer which will be filled with the specified number of items from the PCI BAR space. The allocated space must be large enough to hold the specified amount of data.

## EXAMPLE

```
#include "tdrv020api.h"

#define NUM_ITEMS 256

TDRV020_HANDLE hdl;
TDRV020_STATUS result;
int offset;
unsigned char dataBuf[NUM_ITEMS];

offset = 0x0000;
/*
** read 256 Bytes from the User FPGA Register Area
*/
result = tdrv020Read8(hdl, TDRV020_RES_MEM_1, offset, NUM_ITEMS, dataBuf);

if (result != TDRV020_OK)
{
    /* handle error */
}
```

## RETURN VALUE

On success, TDRV020\_OK is returned. In the case of an error, the appropriate error code is returned by the function.

## ERROR CODES

Error code	Description
TDRV020_ERR_INVALID_HANDLE	The specified device handle is invalid
TDRV020_ERR_INVALID	The specified access range exceeds PCI BAR limits
TDRV020_ERR_ACCESS	The specified PCI resource is not available

## 3.2.2 tdrv020ReadBE16

### NAME

tdrv020ReadBE16 – read 16-bit values from PCI BAR space in big-endian order

### SYNOPSIS

```
TDRV020_STATUS tdrv020ReadBE16
(
    TDRV020_HANDLE    hdl,
    int                pciResource,
    int                offset,
    int                numItems,
    unsigned short     *pData
)
```

### DESCRIPTION

This function reads the specified number of items from the PCI BAR space by using 16-bit accesses. The values are returned as big-endian values that mean on Intel x86 architectures the multi-byte data will be byte-swapped.

### PARAMETERS

*hdl*

This argument specifies the device handle to the hardware module retrieved by a call to the corresponding open-function.

*pciResource*

This parameter specifies the desired PCI Memory resource to be used for this access. In general, a PCI target supports up to six base address registers. Following values are possible:

Value	Description
TDRV020_RES_MEM_1	First found PCI Memory area.
TDRV020_RES_MEM_2	Second found PCI Memory area.
TDRV020_RES_MEM_3	Third found PCI Memory area.
TDRV020_RES_MEM_4	Fourth found PCI Memory area.
TDRV020_RES_MEM_5	Fifth found PCI Memory area.
TDRV020_RES_MEM_6	Sixth found PCI Memory area.

The Base Address Register usage is programmable and can be changed by modifying the PCIe bridge configuration. Therefore the following table is just an example how the PCI Base Address Registers could be used.

PCI Base Address Register	PCI Address-Type	TDRV020 Resource
0	MEM	TDRV020_RES_MEM_1
1	MEM ( <i>not used</i> )	TDRV020_RES_MEM_2
2	MEM ( <i>not used</i> )	TDRV020_RES_MEM_3

*offset*

This argument specifies the start offset within the PCI BAR space.

*numItems*

This argument specifies the number of items (16-bit) to read.

*pData*

This argument is a pointer to an unsigned short buffer which will be filled with the specified number of items from the PCI BAR space. The allocated space must be large enough to hold the specified amount of data.

**EXAMPLE**

```
#include "tdrv020api.h"

#define NUM_ITEMS 128

TDRV020_HANDLE hdl;
TDRV020_STATUS result;
int offset;
unsigned short dataBuf[NUM_ITEMS];

offset = 0x0000;
/*
** read 256 Bytes from the User FPGA Register Area
*/
result = tdrv020ReadBE16(hdl, TDRV020_RES_MEM_1, offset, NUM_ITEMS,
                        dataBuf);

if (result != TDRV020_OK)
{
    /* handle error */
}
```

## RETURN VALUE

On success, TDRV020\_OK is returned. In the case of an error, the appropriate error code is returned by the function.

## ERROR CODES

Error code	Description
TDRV020_ERR_INVALID_HANDLE	The specified device handle is invalid
TDRV020_ERR_INVALID	The specified access range exceeds PCI BAR limits
TDRV020_ERR_ACCESS	The specified PCI resource is not available

### 3.2.3 tdrv020ReadLE16

#### NAME

tdrv020ReadLE16 – read 16-bit values from PCI BAR space in little-endian order

#### SYNOPSIS

```
TDRV020_STATUS tdrv020ReadLE16
(
    TDRV020_HANDLE    hdl,
    int               pciResource,
    int               offset,
    int               numItems,
    unsigned short    *pData
)
```

#### DESCRIPTION

This function reads the specified number of items from the PCI BAR space by using 16-bit accesses. The values are returned as little-endian values that means on Intel x86 architectures the multi-byte data will not be byte-swapped.

#### PARAMETERS

*hdl*

This argument specifies the device handle to the hardware module retrieved by a call to the corresponding open-function.

*pciResource*

This parameter specifies the desired PCI Memory resource to be used for this access. In general, a PCI target supports up to six base address registers. Following values are possible:

Value	Description
TDRV020_RES_MEM_1	First found PCI Memory area.
TDRV020_RES_MEM_2	Second found PCI Memory area.
TDRV020_RES_MEM_3	Third found PCI Memory area.
TDRV020_RES_MEM_4	Fourth found PCI Memory area.
TDRV020_RES_MEM_5	Fifth found PCI Memory area.
TDRV020_RES_MEM_6	Sixth found PCI Memory area.

The Base Address Register usage is programmable and can be changed by modifying the PCIe bridge configuration. Therefore the following table is just an example how the PCI Base Address Registers could be used.

PCI Base Address Register	PCI Address-Type	TDRV020 Resource
0	MEM	TDRV020_RES_MEM_1
1	MEM ( <i>not used</i> )	TDRV020_RES_MEM_2
2	MEM ( <i>not used</i> )	TDRV020_RES_MEM_3

#### *offset*

This argument specifies the start offset within the PCI BAR space.

#### *numItems*

This argument specifies the number of items (16-bit) to read.

#### *pData*

This argument is a pointer to an unsigned short buffer which will be filled with the specified number of items from the PCI BAR space. The allocated space must be large enough to hold the specified amount of data.

## EXAMPLE

```
#include "tdrv020api.h"

#define NUM_ITEMS 128

TDRV020_HANDLE hdl;
TDRV020_STATUS result;
int offset;
unsigned short dataBuf[NUM_ITEMS];

offset = 0x0000;
/*
** read 256 Bytes from the User FPGA Register Area
*/
result = tdrv020ReadLE16(hdl, TDRV020_RES_MEM_1, offset, NUM_ITEMS,
                        dataBuf);

if (result != TDRV020_OK)
{
    /* handle error */
}
```

## RETURN VALUE

On success, TDRV020\_OK is returned. In the case of an error, the appropriate error code is returned by the function.

## ERROR CODES

Error Code	Description
TDRV020_ERR_INVALID_HANDLE	The specified device handle is invalid
TDRV020_ERR_INVALID	The specified access range exceeds PCI BAR limits
TDRV020_ERR_ACCESS	The specified PCI resource is not available



### 3.2.4 tdrv020ReadBE32

#### NAME

tdrv020ReadBE32 – read 32-bit values from PCI BAR space in big-endian order

#### SYNOPSIS

```
TDRV020_STATUS tdrv020ReadBE32
(
    TDRV020_HANDLE    hdl,
    int                pciResource,
    int                offset,
    int                numItems,
    unsigned int       *pData
)
```

#### DESCRIPTION

This function reads the specified number of items from the PCI BAR space by using 32-bit accesses. The values are returned as big-endian values that means on Intel x86 architectures the multi-byte data will be byte-swapped.

#### PARAMETERS

*hdl*

This argument specifies the device handle to the hardware module retrieved by a call to the corresponding open-function.

*pciResource*

This parameter specifies the desired PCI Memory resource to be used for this access. In general, a PCI target supports up to six base address registers. Following values are possible:

Value	Description
TDRV020_RES_MEM_1	First found PCI Memory area.
TDRV020_RES_MEM_2	Second found PCI Memory area.
TDRV020_RES_MEM_3	Third found PCI Memory area.
TDRV020_RES_MEM_4	Fourth found PCI Memory area.
TDRV020_RES_MEM_5	Fifth found PCI Memory area.
TDRV020_RES_MEM_6	Sixth found PCI Memory area.

The Base Address Register usage is programmable and can be changed by modifying the PCIe bridge configuration. Therefore the following table is just an example how the PCI Base Address Registers could be used.

PCI Base Address Register	PCI Address-Type	TDRV020 Resource
0	MEM	TDRV020_RES_MEM_1
1	MEM ( <i>not used</i> )	TDRV020_RES_MEM_2
2	MEM ( <i>not used</i> )	TDRV020_RES_MEM_3

#### *offset*

This argument specifies the start offset within the PCI BAR space.

#### *numItems*

This argument specifies the number of items (32-bit) to read.

#### *pData*

This argument is a pointer to an unsigned int buffer which will be filled with the specified number of items from the PCI BAR space. The allocated space must be large enough to hold the specified amount of data.

## EXAMPLE

```
#include "tdrv020api.h"

#define NUM_ITEMS 1

TDRV020_HANDLE hdl;
TDRV020_STATUS result;
int offset;
unsigned int dataBuf[NUM_ITEMS];

offset = 0;
/*
** read Digital Input Register of FPGA Example Design
*/
result = tdrv020ReadBE32(hdl, TDRV020_RES_MEM_1, offset, NUM_ITEMS,
                        dataBuf);

if (result != TDRV020_OK)
{
    /* handle error */
}
```

## RETURN VALUE

On success, TDRV020\_OK is returned. In the case of an error, the appropriate error code is returned by the function.

## ERROR CODES

Error Code	Description
TDRV020_ERR_INVALID_HANDLE	The specified device handle is invalid
TDRV020_ERR_INVALID	The specified access range exceeds PCI BAR limits
TDRV020_ERR_ACCESS	The specified PCI resource is not available

### 3.2.5 tdrv020ReadLE32

#### NAME

tdrv020ReadLE32 – read 32-bit values from PCI BAR space in little-endian order

#### SYNOPSIS

```
TDRV020_STATUS tdrv020ReadLE32
(
    TDRV020_HANDLE    hdl,
    int               pciResource,
    int               offset,
    int               numItems,
    unsigned int      *pData
)
```

#### DESCRIPTION

This function reads the specified number of items from the PCI BAR space by using 32-bit accesses. The values are returned as little-endian values that means on Intel x86 architectures the multi-byte data will not be byte-swapped.

#### PARAMETERS

*hdl*

This argument specifies the device handle to the hardware module retrieved by a call to the corresponding open-function.

*pciResource*

This parameter specifies the desired PCI Memory resource to be used for this access. In general, a PCI target supports up to six base address registers. Following values are possible:

Value	Description
TDRV020_RES_MEM_1	First found PCI Memory area.
TDRV020_RES_MEM_2	Second found PCI Memory area.
TDRV020_RES_MEM_3	Third found PCI Memory area.
TDRV020_RES_MEM_4	Fourth found PCI Memory area.
TDRV020_RES_MEM_5	Fifth found PCI Memory area.
TDRV020_RES_MEM_6	Sixth found PCI Memory area.

The Base Address Register usage is programmable and can be changed by modifying the PCIe bridge configuration. Therefore the following table is just an example how the PCI Base Address Registers could be used.

PCI Base Address Register	PCI Address-Type	TDRV020 Resource
0	MEM	TDRV020_RES_MEM_1
1	MEM ( <i>not used</i> )	TDRV020_RES_MEM_2
2	MEM ( <i>not used</i> )	TDRV020_RES_MEM_3

*offset*

This argument specifies the start offset within the PCI BAR space.

*numItems*

This argument specifies the number of items (32-bit) to read.

*pData*

This argument is a pointer to an unsigned int buffer which will be filled with the specified number of items from the PCI BAR space. The allocated space must be large enough to hold the specified amount of data.

**EXAMPLE**

```
#include "tdrv020api.h"

#define NUM_ITEMS 1

TDRV020_HANDLE hdl;
TDRV020_STATUS result;
int offset;
unsigned int dataBuf[NUM_ITEMS];

offset = 0;
/*
** read Digital Input Register of FPGA Example Design
*/
result = tdrv020ReadLE32(hdl, TDRV020_RES_MEM_1, offset, NUM_ITEMS,
                        dataBuf);

if (result != TDRV020_OK)
{
    /* handle error */
}
```

## RETURN VALUE

On success, TDRV020\_OK is returned. In the case of an error, the appropriate error code is returned by the function.

## ERROR CODES

Error Code	Description
TDRV020_ERR_INVALID_HANDLE	The specified device handle is invalid
TDRV020_ERR_INVALID	The specified access range exceeds PCI BAR limits
TDRV020_ERR_ACCESS	The specified PCI resource is not available

### 3.2.6 tdrv020Write8

#### NAME

tdrv020Write8 – write 8-bit values to the PCI BAR space

#### SYNOPSIS

```
TDRV020_STATUS tdrv020Write8
(
    TDRV020_HANDLE    hdl,
    int                pciResource,
    int                offset,
    int                numItems,
    unsigned char      *pData
)
```

#### DESCRIPTION

This function writes the specified number of items to the PCI BAR space by using single byte (8-bit) accesses.

#### PARAMETERS

*hdl*

This argument specifies the device handle to the hardware module retrieved by a call to the corresponding open-function.

*pciResource*

This parameter specifies the desired PCI Memory resource to be used for this access. In general, a PCI target supports up to six base address registers. Following values are possible:

Value	Description
TDRV020_RES_MEM_1	First found PCI Memory area.
TDRV020_RES_MEM_2	Second found PCI Memory area.
TDRV020_RES_MEM_3	Third found PCI Memory area.
TDRV020_RES_MEM_4	Fourth found PCI Memory area.
TDRV020_RES_MEM_5	Fifth found PCI Memory area.
TDRV020_RES_MEM_6	Sixth found PCI Memory area.

The Base Address Register usage is programmable and can be changed by modifying the PCIe bridge configuration. Therefore the following table is just an example how the PCI Base Address Registers could be used.

PCI Base Address Register	PCI Address-Type	TDRV020 Resource
0	MEM	TDRV020_RES_MEM_1
1	MEM ( <i>not used</i> )	TDRV020_RES_MEM_2
2	MEM ( <i>not used</i> )	TDRV020_RES_MEM_3

*offset*

This argument specifies the start offset within the PCI BAR space.

*numItems*

This argument specifies the number of items (8-bit) to write.

*pData*

This argument is a pointer to an unsigned char buffer with the data items to write. The allocated space must be large enough to hold the specified amount of data.

**EXAMPLE**

```
#include "tdrv020api.h"

#define NUM_ITEMS 4

TDRV020_HANDLE hdl;
TDRV020_STATUS result;
int offset;
unsigned char dataBuf[NUM_ITEMS];

dataBuf[0] = 0xAA;
dataBuf[1] = 0x55;
...

offset = 0x00;

/*
** write 4 bytes to a 32bit Scratchpad Register
*/
result = tdrv020Write8(hdl, TDRV020_RES_MEM_1, offset, NUM_ITEMS, dataBuf);

if (result != TDRV020_OK)
{
    /* handle error */
}
```



---

## RETURN VALUE

On success, TDRV020\_OK is returned. In the case of an error, the appropriate error code is returned by the function.

## ERROR CODES

Error Code	Description
TDRV020_ERR_INVALID_HANDLE	The specified device handle is invalid
TDRV020_ERR_INVALID	The specified access range exceeds PCI BAR limits
TDRV020_ERR_ACCESS	The specified PCI resource is not available

### 3.2.7 tdrv020WriteBE16

#### NAME

tdrv020WriteBE16 – write 16-bit values to the PCI BAR space big-endian order

#### SYNOPSIS

```
TDRV020_STATUS tdrv020WriteBE16
(
    TDRV020_HANDLE    hdl,
    int                pciResource,
    int                offset,
    int                numItems,
    unsigned short     *pData
)
```

#### DESCRIPTION

This function writes the specified number of items to the PCI BAR space by using 16-bit accesses.

The values are written in big-endian order that means on Intel x86 architectures the multi-byte data will be byte-swapped.

#### PARAMETERS

*hdl*

This argument specifies the device handle to the hardware module retrieved by a call to the corresponding open-function.

*pciResource*

This parameter specifies the desired PCI Memory resource to be used for this access. In general, a PCI target supports up to six base address registers. Following values are possible:

Value	Description
TDRV020_RES_MEM_1	First found PCI Memory area.
TDRV020_RES_MEM_2	Second found PCI Memory area.
TDRV020_RES_MEM_3	Third found PCI Memory area.
TDRV020_RES_MEM_4	Fourth found PCI Memory area.
TDRV020_RES_MEM_5	Fifth found PCI Memory area.
TDRV020_RES_MEM_6	Sixth found PCI Memory area.

The Base Address Register usage is programmable and can be changed by modifying the PCIe bridge configuration. Therefore the following table is just an example how the PCI Base Address Registers could be used.

PCI Base Address Register	PCI Address-Type	TDRV020 Resource
0	MEM	TDRV020_RES_MEM_1
1	MEM ( <i>not used</i> )	TDRV020_RES_MEM_2
2	MEM ( <i>not used</i> )	TDRV020_RES_MEM_3

*offset*

This argument specifies the start offset within the PCI BAR space.

*numItems*

This argument specifies the number of items (16-bit) to write.

*pData*

This argument is a pointer to an unsigned short buffer with the data items to write. The allocated space must be large enough to hold the specified amount of data.

**EXAMPLE**

```
#include "tdrv020api.h"

#define NUM_ITEMS 2

TDRV020_HANDLE hdl;
TDRV020_STATUS result;
int offset;
unsigned short dataBuf[NUM_ITEMS];

dataBuf[0] = 0xAA55;
dataBuf[1] = 0x55AA;
...

offset = 0xF8;
/*
** write 2 datawords to a 32bit Scratchpad Register
*/
result = tdrv020WriteBE16(hdl, TDRV020_RES_MEM_1, offset, NUM_ITEMS,
                          dataBuf);

if (result != TDRV020_OK)
{
    /* handle error */
}
```

## RETURN VALUE

On success, TDRV020\_OK is returned. In the case of an error, the appropriate error code is returned by the function.

## ERROR CODES

Error Code	Description
TDRV020_ERR_INVALID_HANDLE	The specified device handle is invalid
TDRV020_ERR_INVALID	The specified access range exceeds PCI BAR limits
TDRV020_ERR_ACCESS	The specified PCI resource is not available

### 3.2.8 tdrv020WriteLE16

#### NAME

tdrv020WriteLE16 – write 16-bit values to the PCI BAR space in little-endian order

#### SYNOPSIS

```
TDRV020_STATUS tdrv020WriteLE16
(
    TDRV020_HANDLE    hdl,
    int                pciResource,
    int                offset,
    int                numItems,
    unsigned short    *pData
)
```

#### DESCRIPTION

This function writes the specified number of items to the PCI BAR space by using 16-bit accesses.

The values are written in little-endian order that means on Intel x86 architectures the multi-byte data will not be byte-swapped.

#### PARAMETERS

*hdl*

This argument specifies the device handle to the hardware module retrieved by a call to the corresponding open-function.

*pciResource*

This parameter specifies the desired PCI Memory resource to be used for this access. In general, a PCI target supports up to six base address registers. Following values are possible:

Value	Description
TDRV020_RES_MEM_1	First found PCI Memory area.
TDRV020_RES_MEM_2	Second found PCI Memory area.
TDRV020_RES_MEM_3	Third found PCI Memory area.
TDRV020_RES_MEM_4	Fourth found PCI Memory area.
TDRV020_RES_MEM_5	Fifth found PCI Memory area.
TDRV020_RES_MEM_6	Sixth found PCI Memory area.

The Base Address Register usage is programmable and can be changed by modifying the PCIe bridge configuration. Therefore the following table is just an example how the PCI Base Address Registers could be used.

PCI Base Address Register	PCI Address-Type	TDRV020 Resource
0	MEM	TDRV020_RES_MEM_1
1	MEM ( <i>not used</i> )	TDRV020_RES_MEM_2
2	MEM ( <i>not used</i> )	TDRV020_RES_MEM_3

*offset*

This argument specifies the start offset within the PCI BAR space.

*numItems*

This argument specifies the number of items (16-bit) to write.

*pData*

This argument is a pointer to an unsigned short buffer with the data items to write. The allocated space must be large enough to hold the specified amount of data.

## EXAMPLE

```
#include "tdrv020api.h"

#define NUM_ITEMS 2

TDRV020_HANDLE hdl;
TDRV020_STATUS result;
int offset;
unsigned short dataBuf[NUM_ITEMS];

dataBuf[0] = 0xAA55;
dataBuf[1] = 0x55AA;
...

offset = 0xF8;
/*
** write 2 data words to a 32bit Scratchpad Register
*/
result = tdrv020WriteLE16(hdl, TDRV020_RES_MEM_1, offset, NUM_ITEMS,
                          dataBuf);

if (result != TDRV020_OK)
{
    /* handle error */
}
```

---

## RETURN VALUE

On success, TDRV020\_OK is returned. In the case of an error, the appropriate error code is returned by the function.

## ERROR CODES

Error Code	Description
TDRV020_ERR_INVALID_HANDLE	The specified device handle is invalid
TDRV020_ERR_INVALID	The specified access range exceeds PCI BAR limits
TDRV020_ERR_ACCESS	The specified PCI resource is not available

### 3.2.9 tdrv020WriteBE32

#### NAME

tdrv020WriteBE32 – write 32-bit values to the PCI BAR space big-endian order

#### SYNOPSIS

```
TDRV020_STATUS tdrv020WriteBE32
(
    TDRV020_HANDLE    hdl,
    int               pciResource,
    int               offset,
    int               numItems,
    unsigned int      *pData
)
```

#### DESCRIPTION

This function writes the specified number of items to the PCI BAR space by using 32-bit accesses.

The values are written in big-endian order that means on Intel x86 architectures the multi-byte data will be byte-swapped.

#### PARAMETERS

*hdl*

This argument specifies the device handle to the hardware module retrieved by a call to the corresponding open-function.

*pciResource*

This parameter specifies the desired PCI Memory resource to be used for this access. In general, a PCI target supports up to six base address registers. Following values are possible:

Value	Description
TDRV020_RES_MEM_1	First found PCI Memory area.
TDRV020_RES_MEM_2	Second found PCI Memory area.
TDRV020_RES_MEM_3	Third found PCI Memory area.
TDRV020_RES_MEM_4	Fourth found PCI Memory area.
TDRV020_RES_MEM_5	Fifth found PCI Memory area.
TDRV020_RES_MEM_6	Sixth found PCI Memory area.

The Base Address Register usage is programmable and can be changed by modifying the PCIe bridge configuration. Therefore the following table is just an example how the PCI Base Address Registers could be used.



PCI Base Address Register	PCI Address-Type	TDRV020 Resource
0	MEM	TDRV020_RES_MEM_1
1	MEM ( <i>not used</i> )	TDRV020_RES_MEM_2
2	MEM ( <i>not used</i> )	TDRV020_RES_MEM_3

*offset*

This argument specifies the start offset within the PCI BAR space.

*numItems*

This argument specifies the number of items (32-bit) to write.

*pData*

This argument is a pointer to an unsigned int buffer with the data items to write. The allocated space must be large enough to hold the specified amount of data.

## EXAMPLE

```
#include "tdrv020api.h"

TDRV020_HANDLE    hdl;
TDRV020_STATUS    result;
int               offset;
unsigned int       data;

data = 0x12345678;
offset = 0xF8;
/* Write Test data into a 32bit Scratchpad Register */
result = tdrv020WriteBE32(hdl, TDRV020_RES_MEM_1, offset, 1, &data);

if (result != TDRV020_OK)
{
    /* handle error */
}
```

## RETURN VALUE

On success, TDRV020\_OK is returned. In the case of an error, the appropriate error code is returned by the function.

## ERROR CODES

Error Code	Description
TDRV020_ERR_INVALID_HANDLE	The specified device handle is invalid
TDRV020_ERR_INVALID	The specified access range exceeds PCI BAR limits
TDRV020_ERR_ACCESS	The specified PCI resource is not available

## 3.2.10 tdrv020WriteLE32

### NAME

tdrv020WriteLE32 – write 32-bit values to the PCI BAR space in little-endian order

### SYNOPSIS

```
TDRV020_STATUS tdrv020WriteLE32
(
    TDRV020_HANDLE    hdl,
    int               pciResource,
    int               offset,
    int               numItems,
    unsigned int      *pData
)
```

### DESCRIPTION

This function writes the specified number of items to the PCI BAR space by using 32-bit accesses.

The values are written in little-endian order that means on Intel x86 architectures the multi-byte data will not be byte-swapped.

### PARAMETERS

*hdl*

This argument specifies the device handle to the hardware module retrieved by a call to the corresponding open-function.

*pciResource*

This parameter specifies the desired PCI Memory resource to be used for this access. In general, a PCI target supports up to six base address registers. Following values are possible:

Value	Description
TDRV020_RES_MEM_1	First found PCI Memory area.
TDRV020_RES_MEM_2	Second found PCI Memory area.
TDRV020_RES_MEM_3	Third found PCI Memory area.
TDRV020_RES_MEM_4	Fourth found PCI Memory area.
TDRV020_RES_MEM_5	Fifth found PCI Memory area.
TDRV020_RES_MEM_6	Sixth found PCI Memory area.

The Base Address Register usage is programmable and can be changed by modifying the PCIe bridge configuration. Therefore the following table is just an example how the PCI Base Address Registers could be used.

PCI Base Address Register	PCI Address-Type	TDRV020 Resource
0	MEM	TDRV020_RES_MEM_1
1	MEM ( <i>not used</i> )	TDRV020_RES_MEM_2
2	MEM ( <i>not used</i> )	TDRV020_RES_MEM_3

*offset*

This argument specifies the start offset within the PCI BAR space.

*numItems*

This argument specifies the number of items (32-bit) to write.

*pData*

This argument is a pointer to an unsigned int buffer with the data items to write. The allocated space must be large enough to hold the specified amount of data.

## EXAMPLE

```
#include "tdrv020api.h"

TDRV020_HANDLE    hdl;
TDRV020_STATUS    result;
int               offset;
unsigned int       data;

data    = 0x12345678;
offset  = 0xF8;
/* Write Test data into a 32bit Scratchpad Register */
result = tdrv020WriteLE32(hdl, TDRV020_RES_MEM_1, offset, 1, &data);

if (result != TDRV020_OK)
{
    /* handle error */
}
```

## RETURN VALUE

On success, TDRV020\_OK is returned. In the case of an error, the appropriate error code is returned by the function.

## ERROR CODES

Error Code	Description
TDRV020_ERR_INVALID_HANDLE	The specified device handle is invalid
TDRV020_ERR_INVALID	The specified access range exceeds PCI BAR limits
TDRV020_ERR_ACCESS	The specified PCI resource is not available

## 3.3 Resource Mapping Functions

### 3.3.1 tdrv020PciResourceMap

#### NAME

tdrv020PciResourceMap – map a PCI resource directly into the process context

#### SYNOPSIS

```
TDRV020_STATUS tdrv020PciResourceMap
(
    TDRV020_HANDLE    hdl,
    int               pciResource,
    unsigned char     **pPtr,
    unsigned int      *pSize
)
```

#### DESCRIPTION

This function maps the specified PCI resource of the hardware module directly into the process context. The retrieved pointer can be used for direct non-cached register access.

#### PARAMETERS

*hdl*

This argument specifies the device handle to the hardware module retrieved by a call to the corresponding open-function.

*pciResource*

This parameter specifies the desired PCI Memory resource to be used for this access. In general, a PCI target (PCIe bridge) supports up to six base address registers. Following values are possible:

Value	Description
TDRV020_RES_MEM_1	First found PCI Memory area.
TDRV020_RES_MEM_2	Second found PCI Memory area.
TDRV020_RES_MEM_3	Third found PCI Memory area.
TDRV020_RES_MEM_4	Fourth found PCI Memory area.
TDRV020_RES_MEM_5	Fifth found PCI Memory area.
TDRV020_RES_MEM_6	Sixth found PCI Memory area.

The Base Address Register usage is programmable and can be changed by modifying the PCIe bridge configuration. Therefore the following table is just an example how the PCI Base Address Registers could be used.

PCI Base Address Register	PCI Address-Type	TDRV020 Resource
0	MEM	TDRV020_RES_MEM_1
1	MEM ( <i>not used</i> )	TDRV020_RES_MEM_2
2	MEM ( <i>not used</i> )	TDRV020_RES_MEM_3

#### *pPtr*

This argument is a pointer to an unsigned char pointer that receives the start address of the mapped PCI resource.

#### *pSize*

This argument returns the size of the mapped PCI resource in bytes.

## EXAMPLE

```
#include "tdrv020api.h"

TDRV020_HANDLE    hdl;
TDRV020_STATUS    result;
unsigned char     *pReg;
unsigned int      size;

/*
** map first memory PCI resource
*/
result = tdrv020PciResourceMap(hdl, TDRV020_RES_MEM_1, &pReg, &size);

if (result != TDRV020_OK)
{
    /* handle error */
}
```

## RETURN VALUE

On success, TDRV020\_OK is returned. In the case of an error, the appropriate error code is returned by the function.

## ERROR CODES

Error Code	Description
TDRV020_ERR_INVALID_HANDLE	The specified device handle is invalid
TDRV020_ERR_ACCESS	Specified PCI resource not available
TDRV020_ERR_NOMEM	Unable to allocate memory

### 3.3.2 tdrv020PciResourceUnmap

#### NAME

tdrv020PciResourceUnmap – unmap a previously mapped PCI resource

#### SYNOPSIS

```
TDRV020_STATUS tdrv020PciResourceUnmap
(
    TDRV020_HANDLE    hdl,
    unsigned char     *pPtr
)
```

#### DESCRIPTION

This function unmaps a previously mapped PCI resource, freeing the system resources used for this mapping.

#### PARAMETERS

*hdl*

This argument specifies the device handle to the hardware module retrieved by a call to the corresponding open-function.

*pPtr*

This argument is a pointer to an unsigned char pointer that represents the start address of the previously mapped PCI resource. This pointer must have been received from the corresponding mapping function.



## EXAMPLE

```
#include "tdrv020api.h"

TDRV020_HANDLE    hdl;
TDRV020_STATUS    result;
unsigned char     *pReg;

/*
** unmap a previously mapped PCI resource
*/
result = tdrv020PciResourceUnmap(hdl, pReg);

if (result != TDRV020_OK)
{
    /* handle error */
}
```

## RETURN VALUE

On success, TDRV020\_OK is returned. In the case of an error, the appropriate error code is returned by the function.

## ERROR CODES

Error Code	Description
TDRV020_ERR_INVALID_HANDLE	The specified device handle is invalid
TDRV020_ERR_INVAL	Invalid pointer specified

## 3.4 Interrupt Functions

### 3.4.1 tdrv020InterruptConfig

#### NAME

tdrv020InterruptConfig – Configure the Interrupt handling method for User-FPGA implementations

#### SYNOPSIS

```
TDRV020_STATUS tdrv020InterruptConfig
(
    TDRV020_HANDLE    hdl,
    int               ControlType,
    int               ReadPciResource,
    int               ReadAccessWidth,
    unsigned int      ReadOffset,
    unsigned int      ReadMask,
    int               WritePciResource,
    int               WriteAccessWidth,
    unsigned int      WriteOffset,
    unsigned int      WriteMask,
    unsigned int      WriteValue
)
```

#### DESCRIPTION

This function configures the interrupt handling method for User-FPGA specific implementations. Interrupt handling must be implemented on driver-level, so the driver must be configured properly to acknowledge an interrupt of the user-specific FPGA implementation.

**Make sure to configure the interrupt handling before the User-FPGA implementation raises interrupts.**

#### PARAMETERS

*hdl*

This value specifies the callback handle retrieved by a call to the corresponding register-function.

### *IntAckMethod*

This value specifies the interrupt acknowledgement method. Following values are possible:

<b>Value</b>	<b>Description</b>
TDRV020_INTACK_READ	Interrupt is cleared upon reading a status register.
TDRV020_INTACK_READCLEAR	Interrupt is cleared by writing the read register bits into the same register, using the same access width as for the read access.
TDRV020_INTACK_READWRITE	Interrupt is cleared upon writing a static value to a specific register.
TDRV020_INTACK_READWRITEMASK	Interrupt is cleared upon writing a static value to a specific register using a bit-mask, leaving specified original register bits unchanged.

### *ReadPciResource*

This parameter specifies the desired PCI Memory resource to be used for reading the interrupt status. In general, a PCI target supports up to six base address registers. Following values are possible:

<b>Value</b>	<b>Description</b>
TDRV020_RES_MEM_1	First found PCI Memory area.
TDRV020_RES_MEM_2	Second found PCI Memory area.
TDRV020_RES_MEM_3	Third found PCI Memory area.
TDRV020_RES_MEM_4	Fourth found PCI Memory area.
TDRV020_RES_MEM_5	Fifth found PCI Memory area.
TDRV020_RES_MEM_6	Sixth found PCI Memory area.

### *ReadAccessWidth*

This parameter specifies the desired access width for reading the interrupt status. The driver always uses little-endian accesses. Following values are possible:

<b>Value</b>	<b>Description</b>
TDRV020_ACCESSWIDTH_8	A BYTE (8bit) register access is used.
TDRV020_ACCESSWIDTH_16	A WORD (16bit) register access is used.
TDRV020_ACCESSWIDTH_32	A DWORD (32bit) register access is used.

### *ReadOffset*

This argument specifies the register offset within the PCI BAR space used for reading the interrupt status.

### *ReadMask*

This argument specifies the bit-mask used for interrupt detection. This argument can be used to mask-out static register bits for proper support of PCI interrupt sharing.

### *WritePciResource*

This parameter specifies the desired PCI Memory resource to be used for writing a static value. In general, a PCI target supports up to six base address registers. This parameter is not used for IntAckMethod READ and READCLEAR. Following values are possible:

<b>Value</b>	<b>Description</b>
TDRV020_RES_MEM_1	First found PCI Memory area.
TDRV020_RES_MEM_2	Second found PCI Memory area.
TDRV020_RES_MEM_3	Third found PCI Memory area.
TDRV020_RES_MEM_4	Fourth found PCI Memory area.
TDRV020_RES_MEM_5	Fifth found PCI Memory area.
TDRV020_RES_MEM_6	Sixth found PCI Memory area.

### *WriteAccessWidth*

This parameter specifies the desired access width for writing the static value. The driver always uses little-endian accesses. This parameter is not used for IntAckMethod READ and READCLEAR. Following values are possible:

<b>Value</b>	<b>Description</b>
TDRV020_ACCESSWIDTH_8	A BYTE (8bit) register access is used.
TDRV020_ACCESSWIDTH_16	A WORD (16bit) register access is used.
TDRV020_ACCESSWIDTH_32	A DWORD (32bit) register access is used.

### *WriteOffset*

This argument specifies the register offset within the PCI BAR space used for writing the static value. This parameter is not used for IntAckMethod READ and READCLEAR.

### *WriteMask*

This argument specifies the bit-mask used for write access. This argument can be used to mask-out static register bits, changing only the desired ones. Specifying 0x00000000 is not valid. This parameter is not used for IntAckMethod READ and READCLEAR.

### *WriteValue*

This argument specifies the static value to be used for writing. This parameter is not used for IntAckMethod READ and READCLEAR.

## EXAMPLE

```
#include "tdrv020api.h"

TDRV020_HANDLE    hdl;
TDRV020_STATUS    result;

/*
** Example 1:
** Configure the Interrupt Handler to use READCLEAR method
** - InterruptStatus Register in first PCI MemRes, at Offset 0x20
** - Use 32bit accesses
*/
result = tdrv020InterruptConfig( hdl,
    TDRV020_INTACK_READCLEAR,
    TDRV020_RES_MEM_1,          // ReadPciResource
    TDRV020_ACCESSWIDTH_32,    // ReadAccessWidth
    0x20,                      // ReadOffset
    0xFFFFFFFF,               // check all register-bits
    0, 0, 0, 0, 0             // do not use write-
                               // parameters
);
if (result == TDRV020_OK)
{
    / *OK */
} else {
    /* handle error */
}

...
```

```

/*
** Example 2:
** Configure the Interrupt Handler to use READWRITE method
** - InterruptStatus Register at PCI Memory Resource 1, Offset 0x20
** - Use 32bit accesses, check all register-bits
** - disable the Interrupt by writing 0x00000000 to PCI Memory Resource 1,
**   Offset 0x24
*/
result = tdrv020InterruptConfig( hdl,
    TDRV020_INTACK_READWRITE,
    TDRV020_RES_MEM_1,           // ReadPciResource
    TDRV020_ACCESSWIDTH_32,     // ReadAccessWidth
    0x20,                       // ReadOffset
    0xFFFFFFFF,                // ReadMask
    TDRV020_RES_MEM_1,           // WritePciResource
    TDRV020_ACCESSWIDTH_32,     // WriteAccessWidth
    0x24,                       // WriteOffset
    0xFFFFFFFF,                // WriteMask
    0x00000000                  // WriteValue
);
if (result == TDRV020_OK)
{
    / *OK */
} else {
    /* handle error */
}

```

## RETURNS

On success, TDRV020\_OK is returned. In the case of an error, the appropriate error code is returned by the function.

## ERROR CODES

Error Code	Description
TDRV020_ERR_INVALID_HANDLE	The specified handle is invalid.
TDRV020_ERR_INVALID	The specified flags are invalid.
TDRV020_ERR_NOSYS	This function is not supported by the device.

### 3.4.2 tdrv020InterruptWait

#### NAME

tdrv020InterruptWait – Wait for incoming Local Interrupt Source

#### SYNOPSIS

```
TDRV020_STATUS tdrv020InterruptWait
(
    TDRV020_HANDLE    hdl,
    unsigned int      interruptMask,
    unsigned int      *pInterruptOccurred,
    int               timeout
)
```

#### DESCRIPTION

This function enables the specified local interrupt sources, and waits for interrupts on the specified local interrupt sources. After an interrupt has arrived, the corresponding occurred local interrupt source is disabled inside the Infrastructure Module (IM). Multiple functions may wait for the same interrupt source to occur.

**The delay between an incoming interrupt and the return of the described function is system-dependent, and is most likely several microseconds.**

#### PARAMETERS

*hdl*

This value specifies the device handle to the hardware module retrieved by a call to the corresponding open-function.

*interruptMask*

This parameter specifies specific interrupt bits to wait for. The interrupt bits correspond to the Infrastructure Module's "Interrupt Pending Register" bits described in the FDK user manual. Please refer to the hardware user manual for further information on the possible interrupt bits. The function returns if at least one of the specified interrupt sources is detected.

*plInterruptOccurred*

If at least one of the specified interrupt sources occurs, the value is returned through this pointer. The interrupt bits correspond to the Infrastructure Module's "Interrupt Pending Register" bits described in the FDK user manual. Please refer to the hardware user manual for further information on the possible interrupt bits.

*timeout*

This value specifies the timeout in milliseconds the function will wait for the interrupt to arrive. The granularity depends on the operating system. To wait indefinitely, specify -1 as timeout parameter.

**EXAMPLE**

```
#include "tdrv020api.h"

TDRV020_HANDLE    hdl;
TDRV020_STATUS    result;
unsigned int      interruptMask;
unsigned int      interruptOccurred;

/*
** Wait at least 5 seconds for incoming interrupts on UINTP0 and/or UINTP1
*/
interruptMask = (1 << 17) | (1 << 16);
result = tdrv020InterruptWait(    hdl,
                                interruptMask,
                                &interruptOccurred,
                                5000 );
if (result == TDRV020_OK)
{
    /* Interrupt arrived. */
    /* Now acknowledge interrupt source in FPGA logic */
    /* to clear the Local Interrupt Source. */
    /* Use tdrv020Read and tdrv020Write functions for */
    /* register access. */
} else {
    /* handle error */
}
```



## RETURNS

On success, TDRV020\_OK is returned. In the case of an error, the appropriate error code is returned by the function.

## ERROR CODES

Error Code	Description
TDRV020_ERR_INVALID_HANDLE	The specified TDRV020_HANDLE is invalid.
TDRV020_ERR_TIMEOUT	The specified timeout occurred.

### 3.4.3 tdrv020InterruptRegisterCallbackThread

#### NAME

tdrv020InterruptRegisterCallbackThread – Register a User Callback Function for Interrupt Handling

#### SYNOPSIS

```
TDRV020_STATUS tdrv020InterruptRegisterCallbackThread
(
    TDRV020_HANDLE    hdl,
    int               threadPriority,
    int               stackSize,
    unsigned int      interruptMask,
    FUNCINTCALLBACK  callbackFunction,
    void              *funcparam,
    TDRV020_HANDLE    *pCallbackHandle
)
```

#### DESCRIPTION

This function registers a user callback function which is executed after detection of the specified interrupt source. It is possible to register multiple callback functions to one or a set (bit mask) of interrupt sources.

The callback function is executed in a thread context, so using TDRV020 device driver functions and system functions is allowed. The callback function should be kept as short as possible. The specified callback function is executed with the occurred interrupt bits and the specified function parameter as function arguments. Additionally, a status value is passed to the callback function, which reflects the result of the involved API functions.

**The delay between an incoming interrupt and the execution of the callback function is system-dependent, and is most likely several microseconds.**

#### PARAMETERS

*hdl*

This value specifies the device handle to the hardware module retrieved by a call to the corresponding open-function.

### *threadPriority*

This parameter specifies the priority to be used for the callback thread. Possible values are:

Value	Description
TDRV020_PRIORITY_NORMAL	Normal Priority (THREAD_PRIORITY_NORMAL)
TDRV020_PRIORITY_HIGH	High Priority (THREAD_PRIORITY_HIGHEST)
TDRV020_PRIORITY_LOW	Low Priority (THREAD_PRIORITY_LOWEST)

Other values might be possible.

### *stackSize*

This parameter specifies the stack size to be used for the callback thread. The value is specified in bytes.

### *interruptMask*

This parameter specifies specific interrupt bits to wait for. The interrupt bits correspond to the Infrastructure Module's "Interrupt Pending Register" bits described in the FDK user manual. Please refer to the hardware user manual for further information on the possible interrupt bits. The callback function is executed if at least one of the specified interrupt sources occurred.

### *callbackFunction*

This parameter is a function pointer to the user callback function. The callback function pointer is defined as follows:

```
typedef void(*FUNCINTCALLBACK)( TDRV020_HANDLE  hdl,
                                unsigned int      interruptOccurred,
                                void              *param,
                                TDRV020_STATUS   status );
```

### *hdl*

This parameter specifies a device handle which can be used for hardware access or other API functions by the callback function.

### *interruptOccurred*

This parameter is a 32bit value reflecting the occurred interrupts. It is useful if the callback function handles multiple interrupt sources. The interrupt bits correspond to the Infrastructure Module's "Interrupt Pending Register" bits described in the FDK user manual. Please refer to the hardware user manual for further information on the possible interrupt bits.

### *param*

This parameter is the user-specified *funcparam* value (see below) which has been specified on callback registration. This value can be used to pass a pointer to a specific control structure, to supply the callback function with specific information.

### *status*

This parameter hands over interrupt callback status information. The callback function needs to check this parameter. If the specified interrupt source has occurred properly, and no errors were detected, this parameter is TDRV020\_OK. If this parameter differs from TDRV020\_OK, an internal error has been detected and the callback handling is stopped. The callback function must implement an appropriate error handling.

*funcparam*

This value specifies a user parameter, which will be handed over to the callback function on execution. This parameter can be used to pass a pointer to a specific control structure used by the callback function.

*pCallbackHandle*

This value specifies a pointer to a handle, where the callback handle will be returned. This callback handle must be used to unregister a callback function.

**EXAMPLE**

```
#include "tdrv020api.h"

TDRV020_HANDLE    hdl;
TDRV020_STATUS    result;
unsigned int      interruptMask;
USER_DATA_AREA    userDataArea;
TDRV020_HANDLE    callbackHandle;

/* forward declaration of callback functions */
void callback_TIMER0(    TDRV020_HANDLE    hdl,
                        unsigned int      interruptOccurred,
                        void               *param,
                        TDRV020_STATUS    status);

/*
** Register callback function for TIMER0 (UINTP0)
** Use a "normal" priority, and 64KB stack.
*/
interruptMask = (1 << 16);
result = tdrv020InterruptRegisterCallbackThread(hdl,
                                                TDRV020_PRIORITY_NORMAL,
                                                0x10000,
                                                interruptMask,
                                                callback_TIMER0,
                                                &userDataArea,
                                                &callbackHandle);

...

```

```
...
if (result != TDRV020_OK)
{
    /* handle error */
}

/*
** Initialize and start the Timer function, using register accesses.
** Refer to the FDK documentation for register description.
*/
...
/*
** Callback Function, using API Functions for Register Access
*/
void callback_TIMER0(    TDRV020_HANDLE    hdl,
                        unsigned int      interruptOccurred,
                        void               *param,
                        TDRV020_STATUS    status)
{
    TDRV020_STATUS    result;
    USER_DATA_AREA    *pUsrData = (USER_DATA_AREA*)param;
    unsigned int      u32value;

    if (status != TDRV020_OK)
    {
        /* handle error status */
    }

    printf("[Timer 0 Interrupt]\n");

    /* Acknowledge TIMER0 interrupt source by writing to
    ** "Timer Based Interrupt Status Register" (offset may differ).
    */
    u32value = (1 << 0);
    result = tdrv020WriteBE32(hdl,
                              TDRV020_RES_MEM_1,
                              0x1002C,
                              1,
                              &u32value );

    /* handle errors */
    return;
}
```

## RETURNS

On success, TDRV020\_OK is returned. In the case of an error, the appropriate error code is returned by the function.

## ERROR CODES

Error Code	Description
TDRV020_ERR_INVALID_HANDLE	The specified TDRV020_HANDLE is invalid.
TDRV020_ERR_INVALID	Function or callback handle pointer is NULL.
TDRV020_ERR_TASK_CREATE	Creation of the callback thread (task) failed.

### 3.4.4 tdrv020InterruptUnregisterCallback

#### NAME

tdrv020InterruptUnregisterCallback – Unregister a User Callback Function

#### SYNOPSIS

```
TDRV020_STATUS tdrv020InterruptUnregisterCallback
(
    TDRV020_HANDLE    hdl
)
```

#### DESCRIPTION

This function unregisters a previously registered user callback thread or ISR function.

#### PARAMETERS

*hdl*

This value specifies the callback handle retrieved by a call to the corresponding register-function.

#### EXAMPLE

```
#include "tdrv020api.h"

TDRV020_HANDLE    callbackHdl;
TDRV020_STATUS    result;

/*
** Unregister a callback function
*/
result = tdrv020InterruptUnregisterCallback(callbackHdl);

if (result == TDRV020_OK)
{
    / *OK */
} else {
    /* handle error */
}
```

## RETURNS

On success, TDRV020\_OK is returned. In the case of an error, the appropriate error code is returned by the function.

## ERROR CODES

Error Code	Description
TDRV020_ERR_INVALID_HANDLE	The specified callback handle is invalid.



## 3.5 Endian Conversion Functions

The following conversion functions can be used to develop endian-neutral software, especially for direct access to mapped PCI resources.

### 3.5.1 endian\_be16

#### NAME

endian\_be16 – big-endian conversion function

#### SYNOPSIS

```
unsigned short endian_be16
(
    unsigned short    u16value
)
```

#### DESCRIPTION

This function converts a short integer value (16-bit) from the native CPU endian order to big-endian order. That means on Intel x86 architectures the value will be byte-swapped, as opposed to PowerPC architectures.

#### PARAMETERS

*u16value*

This argument specifies the data to convert

#### EXAMPLE

```
#include "tdrv020api.h"

unsigned short *pRawData, bigEndianData;

/* setup pRawData pointer to the correct location first */

bigEndianData = endian_be16(*pRawData);
```

#### RETURN VALUE

This function returns the passed value in the big-endian order.

## 3.5.2 endian\_le16

### NAME

endian\_le16 – little-endian conversion function

### SYNOPSIS

```
unsigned short endian_le16
(
    unsigned short    u16value
)
```

### DESCRIPTION

This function converts a short integer value (16-bit) from the native CPU endian order to little-endian order. That means on PowerPC architectures the value will be byte-swapped, as opposed to Intel x86 architectures.

### PARAMETERS

*u16value*

This argument specifies the data to convert

### EXAMPLE

```
#include "tdrv020api.h"

unsigned short *pRawData, littleEndianData;

/* setup pRawData pointer to the correct location first */

littleEndianData = endian_le16(*pRawData);
```

### RETURN VALUE

This function returns the passed value in the little-endian order.

### 3.5.3 endian\_be32

#### NAME

endian\_be32 – big-endian conversion function

#### SYNOPSIS

```
unsigned int endian_be32
(
    unsigned short  u32value
)
```

#### DESCRIPTION

This function converts an integer value (32-bit) from the native CPU endian order to big-endian order. That means on Intel x86 architectures the value will be byte-swapped, as opposed to PowerPC architectures.

#### PARAMETERS

*u32value*

This argument specifies the data to convert

#### EXAMPLE

```
#include "tdrv020api.h"

unsigned short *pRawData, bigEndianData;

/* setup pRawData pointer to the correct location first */

bigEndianData = endian_be32(*pRawData);
```

#### RETURN VALUE

This function returns the passed value in the big-endian order.

### 3.5.4 endian\_le32

#### NAME

endian\_le32 – little-endian conversion function

#### SYNOPSIS

```
unsigned short endian_le32
(
    unsigned short    u32value
)
```

#### DESCRIPTION

This function converts an integer value (32-bit) from the native CPU endian order to little-endian order. That means on PowerPC architectures the value will be byte-swapped, as opposed to Intel x86 architectures.

#### PARAMETERS

*u32value*

This argument specifies the data to convert

#### EXAMPLE

```
#include "tdrv020api.h"

unsigned short *pRawData, littleEndianData;

/* setup pRawData pointer to the correct location first */

littleEndianData = endian_le32(*pRawData);
```

#### RETURN VALUE

This function returns the passed value in the little-endian order.

## 4 Diagnostic

If the TDRV020 does not work properly it is helpful to get some status information from the driver respective kernel.

The Linux */proc* file system provides information about kernel, resources, driver, devices, and so on. The following screen dumps displays information of a correct running TDRV020 driver (see also the *proc* man pages).

```
# lspci -v
...
03:00.0 Signal processing controller: TEWS Technologies GmbH Device a273
  Subsystem: TEWS Technologies GmbH Device a00a
  Flags: bus master, fast devsel, latency 0, IRQ 5
  Memory at febfcc00 (32-bit, non-prefetchable) [size=256]
  Capabilities: [40] Power Management version 3
  Capabilities: [48] MSI: Enable- Count=1/1 Maskable- 64bit+
  Capabilities: [60] Express Endpoint, MSI 00
  Capabilities: [100] Device Serial Number 00-00-00-00-12-34-56-78
...

# cat /proc/devices
Character devices:
  1 mem
  ...
250 tdrv020drv
  ...

# cat /proc/iomem
00000000-0000ffff : reserved
00010000-0009fbff : System RAM
...
feb00000-febffffff : PCI Bus 0000:03
  febfcc00-febfccff : 0000:03:00.0
    febfcc00-febfccff : TDRV020
...
```